Using Reinforce Learning to Solve the Shortest Path Problem

1st Qingyuan FAN
School of Microelectronic
Southern University of Science and Technology
Shenzhen, China
12232509@mail.sustech.edu.cn

***Abstract*—In this peoject, we explored various DP and RL algorithms, including Floyd-Warshall, Q-Learning, Deep Q Networks (DQN), and Greedy algorithms, to solve the Shortest Path Problem (SPP). Dynamic Programming (DP) algorithms such as Floyd-Warshall consistently provided the best results in terms of path length, but these come with significant computational and memory demands, especially for larger graphs.**

***Index Terms*—Dynamic Programming, Reinforce Learning, Q-Learning, Deep Q Network, Greedy**

I. Introduction

The task of shortest path planning is pivotal in a wide variety of applications, ranging from network routing [1] to autonomous vehicle navigation [2]. Traditional Dynamic Programming (DP) methods, such as the Floyd-Warshall algorithm, can indeed determine optimal solutions, but they often prove to be excessively time-consuming when dealing with large graphs. In this light, Reinforcement Learning (RL) provides a balanced solution, offering near-optimal solutions in a more time-efficient manner.

The central goal of this project is to harness different RL and DP algorithms to find the shortest path from any given starting node to any given target node in a weighted directed graph. In our pursuit, we utilize the DP approach (via the Floyd-Warshall algorithm), Q-Learning, and Deep Q Network (DQN) to tackle the problem.

Additionally, we experimented with a bidirectional search strategy involving two agents who simultaneously start searching from the requested end and start nodes to further optimize the path.

Post-optimization, the Q-Learning and DQN models have displayed substantial optimization, capable of controlling the accumulated path (attempting to plan for all pairs of points and summing the path walked by the agent) within five times in large graphs possessing 300 nodes. This is a significant improvement, especially when compared to the greedy algorithm, which used 10 times the accumulated path.

II. Generate the Graph for Dynamic Programming and RL

We first generate the adjacency matrix for the graph. Specifically, value(a,b) of the matrix is the weight from node a to node b, edges that not connected are assigned by zero.

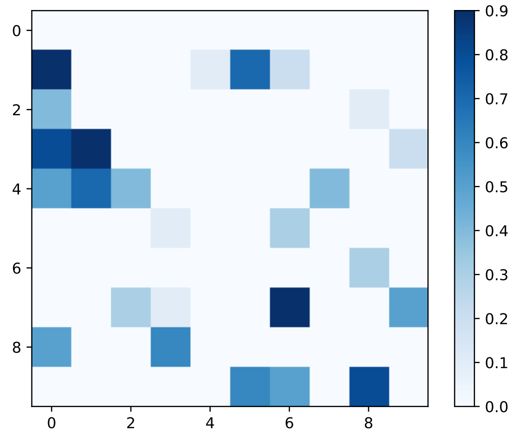

Fig. 1. An adjacency matrix with 10 nodes.

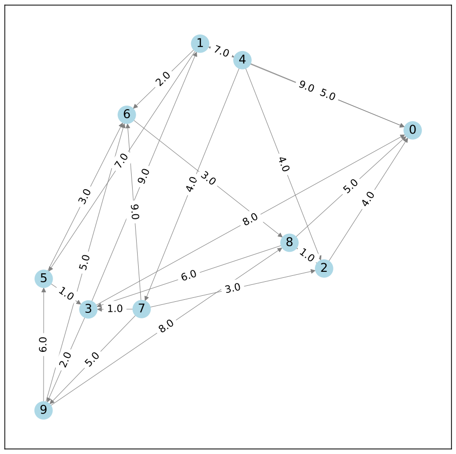

Fig. 2. An graph generate by the adjacency matrix in Figure 1

Algorithm 1 Fill in the adjacency matrix

```
procedure FillAdjacencyMatrix($numNodes, numEdge$)
    $adjacencyMatrix \leftarrow numNodes$ by $numNodes$ matrix filled with zeros
    for $i$ in range 0 to $numEdge - 1$ do
        $node1 \leftarrow$ random number between 0 and $numNodes - 1$
        $node2 \leftarrow$ random number between 0 and $numNodes - 1$
        if $node1 \neq node2$ and $adjacencyMatrix[node1, node2] < 1$ then
            Increment $adjacencyMatrix[node1, node2]$ by a random number between 1 and 10
        end if
    end for
    return $adjacencyMatrix$
end procedure
```

Then, A helper function create_weighted_directed_graph_from_adj_matrix() is defined to transform an adjacency matrix into a directed graph, using the NetworkX library. This function creates an empty directed graph G, adds the specified number of nodes, and then iterates over the adjacency matrix, adding an edge between each pair of nodes if the weight at their intersection in the matrix is greater than zero.

Algorithm 2 Create a weighted directed graph from adjacency matrix

```
procedure CreateWeightedDirectedGraph($adjacencyMatrix$)
    $G \leftarrow$ empty directed graph
    $numNodes \leftarrow$ number of nodes in $adjacencyMatrix$
    for $i$ in range 0 to $numNodes - 1$ do
        Add node $i$ to $G$
    end for
    for $i$ in range 0 to $numNodes - 1$ do
        for $j$ in range 0 to $numNodes - 1$ do
            if $adjacencyMatrix[i, j] > 0$ then
                Add edge from $i$ to $j$ in $G$ with weight $adjacencyMatrix[i, j]$
            end if
        end for
    end for
    return $G$
end procedure
```

III. Algorithm

A. Dynamic Programming

Dynamic programming (DP) is a popular method in graph search algorithms, especially useful when we have a well-defined optimization problem. The Floyd-Warshall algorithm is an example of DP, developed to solve the Shortest Path Problem (SPP). It computes the shortest paths between all pairs of nodes in a weighted, directed graph.

The choice of this algorithm in my project is motivated by the project's requirement: "The solution should still work without re-doing dynamic programming when changing the starting node and target node." The Floyd-Warshall algorithm pre-computes and stores all the shortest paths, allowing for efficient subsequent queries without needing to re-run the entire process.

The logic behind Floyd-Warshall algorithm is as follows:

1) Initialize a distance matrix, where the value at each cell represents the distance between two nodes. The diagonal is filled with zeros, as the distance from a node to itself is zero. For all pairs of nodes that share an edge, their corresponding cell is populated with the weight of that edge.
2) The core of the algorithm involves three nested loops iterating over each node in the graph. For each triplet of nodes (i, j, k), if the sum of the distances from i to k and from k to j is less than the current recorded distance from i to j, we update the distance and set the predecessor of j as the predecessor of k.
3) Initialize a predecessor matrix that will keep track of the path between nodes. For all pairs of nodes that share an edge, the predecessor of the second node is set as the first node.

Algorithm 3 Floyd-Warshall algorithm and Path Reconstruction

```
procedure FloydWarshall($G$)
    $n \leftarrow$ len($G.nodes$)
    $dist \leftarrow$ np.full($(n, n)$, np.inf)
    $pred \leftarrow$ np.full($(n, n), None$)
    np.fill_diagonal($dist, 0$)
    for each edge $(u, v, d)$ in $G.edges(data = True)$ do
        $dist[u, v] \leftarrow d$['weight']
        $pred[u, v] \leftarrow u$
    end for
    for $k \leftarrow 0$ to $n$ do
        for $i \leftarrow 0$ to $n$ do
            for $j \leftarrow 0$ to $n$ do
                if $dist[i, k] + dist[k, j] < dist[i, j]$ then
                    $dist[i, j] \leftarrow dist[i, k] + dist[k, j]$
                    $pred[i, j] \leftarrow pred[k, j]$
                end if
            end for
        end for
    end for
    return $pred, dist$
end procedure
procedure ReconstructPath($u, v, pred$)
    $path \leftarrow$ []
    while $v$ is not None do
        path.append($v$)
        $v \leftarrow pred[u, v]$
    end while
    path.reverse()
    return $path$
end procedure
```

B. Q-Learning

In this section, we use Q-learning as the approach to solve the shortest path problem (SPP). Q-learning is a model-free reinforcement learning algorithm that seeks to find the best action to take given the current state. We have chosen Q-learning as our method of reinforcement learning for solving this problem due to its distinct advantages over other RL algorithms, such as its ability to handle problems with stochastic transitions and rewards, without requiring adaptations.

Q-learning seeks to learn a policy that maximizes the total reward. The Q-learning algorithm iteratively updates the Q-values for each state-action pair using the Bellman equation:

$$Q(s,a) \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s',a')) \quad (1)$$

where s is the current state, r is the immediate reward, s' is the new state, a' is the action taken in the new state, α is the learning rate, γ is the discount factor.

The learning process continues until the Q-values converge to the optimal action-value function. Once this function is learned, the agent can derive the optimal policy by taking the action that maximizes the Q-value in each state. In our program, we use a QLearningAgent class with multiple function to implement the algorithm, whose algorithm is described at Alogrithm 4.

Algorithm 4 Q-Learning Agent

```
1: procedure QLEARNINGAGENT($graph, num\_nodes, \alpha, \gamma, \epsilon$)
2:     Initialize $Q(s, a)$ arbitrarily for all state-action pairs
3:     for episode = 1, $M$ do
4:         Initialize state $s$
5:         for step = 1, $T$ do
6:             Choose action $a$ from state $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy), default $\epsilon$ is 0
7:             Take action $a$, observe reward $r$, next state $s'$
8:             if $s'$ is terminal then
9:                 $Q(s, a) \leftarrow Q(s, a) + \alpha[r - Q(s, a)]$
10:            else
11:                $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
12:            end if
13:            $s \leftarrow s'$
14:        end for
15:    end for
16:    return $Q$
17: end procedure
18: procedure GETOPTIMALPATH($start\_node, end\_node, Q$)
19:    Initialize path with $start\_node$
20:    Initialize $state \leftarrow start\_node$
21:    while $state \neq end\_node$ do
22:        Choose $action$ maximizing $Q(state, action)$ among feasible actions
23:        $state \leftarrow action$
24:        Add $state$ to path
25:    end while
26:    return path
27: end procedure
```

Compared to dynamic programming, Q-learning has a significant advantage in terms of computational complexity, especially when the learning is complete. In dynamic programming, finding the optimal path requires solving the Bellman equation iteratively until convergence, which has a time complexity of $O(V^3)$, where V is the number of vertices in the graph. Q-learning could also deal with the graph that could be changed on the learning process, which is impossible in DP.

On the other hand, once the Q-learning process is complete, the Q-table contains the expected reward for each action in each state. To find the shortest path, the agent only needs to choose the action with the highest expected reward in each state, leading to a path discovery time complexity of $O(V)$, which is significantly lower than that of dynamic programming. This makes Q-learning an effective and efficient method for solving the SPP, particularly in larger graphs.

C. Bidirectional Q-Learning

Bidirectional Q-Learning leverages the fundamental logic of Q-Learning while incorporating the information about the start and end nodes into the learning process. Traditional Q-Learning predominantly employs random walks during training to glean features about the graph, omitting the essential information about the starting and ending nodes as the agent only use the start node to find the q-table and walk until it reach the end node. This approach can be inefficient, particularly when dealing with large graphs, as it necessitates significant exploration before it can provide an optimal solution.

The Bidirectional Q-Learning algorithm, however, innovatively tackles this issue by concurrently initiating searches from both the start and end nodes. This technique inherently halves the search space, which can drastically reduce the computational complexity and expedite the learning process, especially in large graphs. Upon completion of the concurrent searches, the algorithm identifies the intersection points and constructs a potential path based on these intersections. It then evaluates all possible paths to select the one with the best reward, leading to a more efficient, effective, and targeted search for the shortest path.

Algorithm 5 Bidirectional Q-learning

```
1: procedure GET_OPTIMAL_PATH_BIDIRECTIONAL($agent, agent\_R, start\_node, end\_node$)
2:     $path_1 \leftarrow agent.get\_optimal\_path(start\_node, end\_node)$
3:     $path_2 \leftarrow agent\_R.get\_optimal\_path(end\_node, start\_node)$
4:     $path_2 \leftarrow reverse(path\_2)$
5:     $path_1\_set \leftarrow to\_set(path\_1)$
6:     $best\_reward \leftarrow \infty$
7:     $best\_path \leftarrow None$
8:     for each $node$ in $path_2$ do
9:         if $node$ in $path_1\_set$ then
10:            $i \leftarrow index\_of(path_1, node)$
11:            $possible\_path \leftarrow concatenate(path_1[: i], path_2[node :])$
12:            $reward \leftarrow cal\_reward(possible\_path)$
13:            if $reward < best\_reward$ then
14:                $best\_reward \leftarrow reward$
15:                $best\_path \leftarrow possible\_path$
16:            end if
17:        end if
18:    end for
19:    return $best\_path, best\_reward$
20: end procedure
```

D. Value Function Approximation (DQN)

We also tried the approach of value function approximation (VFA) to solve the Shortest Path Problem (SPP). The transition to VFA was informed by the successes realized with the Deep Q-Learning (DQN) model, which is a combination of Q-Learning and Deep Learning.

Deep Q-Learning is an evolution of Q-Learning that employs the use of deep neural networks to approximate the Q-value function. Essentially, DQN is an application of VFA where the value function is approximated by a deep neural network. The key idea is to use a neural network to predict Q-values rather than storing them in a table, thereby making it possible to deal with environments with a large number of states and actions.

The superiority of DQN over traditional Q-Learning can be attributed to several factors. Firstly, DQN is much more scalable as it can handle problems with high dimensional state and action spaces effectively. Secondly, DQN is less prone to overfitting because of its ability to generalize from the input state. Lastly, DQN can handle continuous as well as discrete action spaces, thereby offering more flexibility.

The core algorithm of DQN can be summarized with the following update rule, where Q represents the approximated Q-value function and α is the learning rate:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2}$$

In this equation, s_t is the current state, a_t is the action taken, r_{t+1} is the reward received after taking action a_t at state s_t, and γ is the discount factor.

We use the MLP to implement the DQN in our approach. The architecture of our chosen deep Q-network (QNet) consists of three fully connected layers, with 128 and 64 neurons in the first and second hidden layers, respectively. These layers employ the ReLU activation function. The input of the network is a one-dimensional vector whose length is 3 times the number of nodes. Which contains the current state, next state and the current neighbour of the node to enforce the spatial perception of the agent in the graph. The output of the network is a single neuron representing the approximated Q-value. To prevent overfitting, Layer Normalization and Dropout are applied after the activation function of each hidden layer.

This QNet model was trained using experiences stored in a replay buffer, which provides the benefit of decorrelating the experiences, leading to more stable training. This was further enhanced by employing a target network for more stable and efficient learning.

We also use the technique of experience replay in the training of our network. Experience Replay is an integral part of training our Deep Q-Network. It is essentially a way to store and reuse past experiences, much like how human memory works. Experience Replay allows our network to learn from past experiences, thereby improving its performance.

In our implementation, we define a ReplayBuffer class, which is used to store state transitions. Each transition is a tuple containing the current state, action taken, reward obtained, next state, and next action. The buffer has a fixed size, and once it is filled, new experiences overwrite the old ones. This buffer acts as a memory that the agent uses to recall past experiences. The ReplayBuffer is used during the learning phase of our DQN. After choosing an action using the epsilon-greedy strategy and observing the reward and the new state, we store this transition in our replay buffer. Once enough experiences are collected in the buffer, we randomly sample a batch of transitions and use them to update our network weights. The pseudocode of our algorithm can be found in Algorithm 6

Algorithm 6 Reinforcement Learning Agent with Value Function Approximation and Experience Replay

```
1: procedure INITIALIZATION
2:     Initialize action-value function Q with random weights $\theta$
3:     Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
4:     Initialize experience replay memory D with capacity N
5: end procedure
6: procedure ALGORITHM
7:     for episode = 1, M do
8:         Initialize sequence $s_1$ and preprocessed sequence $\phi_1$
9:         for t = 1, T do
10:            With probability $\epsilon$ select a random action $a_t$
11:            otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
12:            Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
13:            Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
14:            Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
15:            Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
16:            Set $y_j = r_j$ for terminal $\phi_{j+1}$
17:            Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$ for non-terminal $\phi_{j+1}$
18:            Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect
    to the network parameters $\theta$
19:            Every C steps reset $\hat{Q} = Q$
20:        end for
21:    end for
22: end procedure
23: procedure GET_OPTIMAL_PATH(start_node, end_node)
24:    With the learned Q network, compute path from $start\_node$ to $end\_node$
    maximizing the sum of Q values
25:    Return this path
26: end procedure
```

E. Greedy Algorithm

In addition to the previously discussed algorithms, we also implemented a Greedy Algorithm for the purpose of performance comparison with Dynamic Programming and Reinforced Learning.

The Greedy Algorithm is a straightforward yet powerful method that often provides a feasible, if not optimal, solution for many optimization problems. This algorithm operates by making the locally optimal choice at each stage with the hope of finding the global optimum.

In the context of our shortest path problem, the Greedy Algorithm follows the logic of choosing the shortest edge at each step. Starting from the initial node, it inspects all outgoing edges and selects the one with the smallest weight that leads to a node that has not been visited yet. The algorithm proceeds to the next node via this chosen edge and continues this process until it reaches the destination node.

One point to note is that the algorithm has a backtracking mechanism. If it reaches a node with no unvisited neighbors, it will backtrack to the previous node and attempt to find another path. This ensures that the algorithm can handle situations where the immediate choice leads to a dead end and avoids getting stuck. The algorithm of the greedy can be found at Algorithm 7.

Algorithm 7 Greedy Algorithm

```
1: procedure GREEDYPATH($G, start\_node, end\_node$)
2:     $path \leftarrow [start\_node]$
3:     $visited \leftarrow \{start\_node\}$
4:     $state \leftarrow start\_node$
5:     while $state \neq end\_node$ do
6:         $neighbors \leftarrow N(state) \setminus visited$
7:         if $neighbors = \emptyset$ then
8:             $visited.add(state)$
9:             $path.pop()$
10:            $state \leftarrow path[-1]$
11:        else
12:            $next\_node \leftarrow argminlim_{n \in neighbors} w(state, n)$
13:            $visited.add(next\_node)$
14:            $state \leftarrow next\_node$
15:            $path.append(state)$
16:        end if
17:    end while
18:    return $path$
19: end procedure
```

F. Apply RL on Graph with Unknown Node and Edges

The application of Reinforcement Learning (RL) to Graph-based problems, where nodes and edges are unknown, presents a significant challenge. Traditional RL techniques such as Q-Learning and Deep Q Networks (DQN) usually maintain a Q-table, a lookup table for the Q-values associated with each state-action pair. The Q-value quantifies the expected reward of an action taken in a certain state, considering future rewards.

However, when the graph's nodes and weights are not known in advance, it becomes impossible to create this Q-table. We simply do not have the necessary information to construct the state-action space. For example, in Q-Learning, we would typically initialize our Q-table with arbitrary values and then refine them as the agent explores the environment. Without a known environment (or graph, in our case), we cannot initialize such a table. Similarly, DQNs, which abstract the Q-table into a neural network's weights, also suffer from the same limitation. They cannot function without a predefined and structured input, which is not available in our case.

In such scenarios, it turns out that simpler algorithms can be more effective, such as the Greedy algorithm and the bi-directional search mentioned in the previous section. The Greedy algorithm doesn't require a complete knowledge of the environment but incrementally constructs the solution based on the best option at the current moment. This approach lends itself naturally to problems with unknown graphs as the agent can make decisions based solely on its immediate, known surroundings.

Further improvements can be achieved by using a bidirectional Greedy algorithm. In this strategy, two agents are deployed simultaneously, one from the start node and another from the end node. Both agents independently search the graph using the Greedy approach, and the algorithm selects the shortest path found between the two. This bidirectional approach drastically increases the search efficiency as the search space is effectively halved.

In our experiments, we found that the accumulated reward (the sum of the weights of the traversed edges) using the bidirectional Greedy algorithm stays within the same order of magnitude as the optimal solution provided by Dynamic Programming (DP) algorithms. Therefore, in situations where the nodes and edges of the graph are not known in advance, the Greedy and Bidirectional Greedy algorithms offer practical and effective solutions.

IV. Evaluation of the Algorithm

In this section, We aim to investigate the performance of these methods on graphs with different sizes (number of nodes). The algorithms involved are Floyd-Warshall (FW), Q-learning (QL), Deep Q-Network (DQN), Bidirectional Q-learning (Bidirectional Q-Learning), and Greedy search at a graph with deterministic node number and edge.

When assessing the performance of shortest path search algorithms, including reinforcement learning (RL) algorithms and dynamic programming (DP) algorithms, there are several factors we could consider.

1) Efficiency: measure of the computational resources used by an algorithm.
2) Optimality: measure of how good the solutions found by the algorithm are.
3) Scalability: measure of how well the algorithm performs as the problem size increases.
4) Robustness: algorithm's ability to handle a wide range of problem instances.
5) Stability: how much the solutions found by the algorithm vary with small changes in the input data.
6) Convergence: refers to the ability of the algorithm to reach a solution.
7) Sample: refers to the number of samples (or trials) the algorithm needs to find the optimal policy.

However, since our RL algorithm cannot achieve optimum solution in most cases, we mainly analyze the Optimality achieved by different algorithms through the indicators of "accumulated reward " mention in the following paragraph.

Node Number of Graph	Q-Learning	DQN	Bidirectional Q-learning	Greedy
10	1.93	3.57	1.15	1.95
20	3.82	5.46	1.33	3.71
30	6.90	10.70	1.41	5.62
40	12.01	17.62	2.09	7.06
50	13.18	26.11	2.21	9.72
75	21.70	45.35	2.63	13.56
100	25.61	68.09	3.19	18.55
150	34.05		3.59	26.57
200	43.33		3.50	36.36
300	64.08		4.84	56.35
500	104.10		14.09	98.18

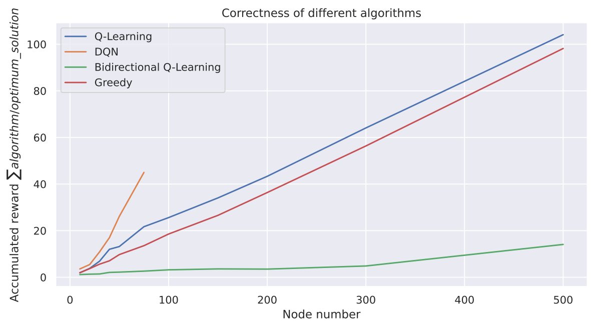

Fig. 3. Performance Comparison of Different Path Planning and RL Algorithms. The figure presents a comparative study of different path planning and reinforcement learning (RL) algorithms, including Floyd-Warshall (FW), Q-learning (QL), Bidirectional Q-learning (Bidirectional Q-Learning), DQN, and a Greedy approach. Each algorithm's performance is tested on graphs of different sizes (indicated by the node numbers). The performance is measured by the fraction of total cost (negative value indicates the cost) from the start node to the goal node of other algorithm compare to Floyd-Warshall. Floyd-Warshall, which provides an optimal solution, serves as the baseline (1) at different node number. As node numbers increase, we can observe the impact on performance for each algorithm.

Our performance evaluation uses the accumulated weight of the edges traversed by the agent during the search divide by the accumulated weight of Floyd-Warshall (consider the DP as the optimum solution and the baseline). For each algorithm, we run the pathfinding process for all pairs of nodes in the graph and obtain the traveled path's total cost. This provides us with a metric to analyze the algorithms' performance against each other.

As the number of nodes in the input graph increases, we observe that the performance of different search/RL methods differs significantly. Classical planning algorithms, such as FW, show consistent performance while RL methods, including Q-learning and DQN, demonstrate a performance degradation when handling larger graphs.

A. Performance of RL versus DP

We found that the performance of the RL agent are all worse than the DP algorithm, which is within our expectations. Because the DP algorithm can always provide the optimal solution, while the RL algorithm is only based on empirical learning, and it takes a long time to train to converge.

B. DQN's Inferior Performance Compared to Q-learning

The performance of the DQN are even worse than Q-learning. Which may caused by the structure of the valur function appoximator of the DQN, who uses a neural network to approximate Q values, which can introduce errors and lead to suboptimal policy choices. Another critical issue of DQN on the SPP question is that the overall optimization process of DQN is conducted with a batch size of 1 (because the agent could not do batch/parallel update), which may lead to unstable training and can result in slower convergence compared to QL. Besides, for these specific pathfinding problems, the nonlinear approximation of Q values resulting from the DQN model may not have brought noticeable benefits, and QL's tabular approach might already be adequate.

C. Bidirectional Q-learning's Superior Performance

Bidirectional Q-Learning displays remarkable performance compared to other RL methods. Its performance is on a similar scale to DP algorithms, while maintaining a complexity comparable to QL. We assumed that this may firstly caused by the algorithm of bidirectional Q-learning, as the bidirectional search simultaneously explores both the forward and backward paths, which can reduce search depth and speed up convergence. Secondly, the interaction between the two search directions reduces the probability of falling into local optima, which is often a concern in classical Q-learning techniques. Lastly, Bidirectional Q-Learning better balances exploration and exploitation trade-offs, maximizing knowledge gain while minimizing redundant exploration.

D. Adaptability of RL Algorithms to SPP Problems

The performance of Q-learning (QL) methods in pathfinding problems appears to be generally consistent with greedy algorithms. We guess this may caused by the following reasons:

1) Difficulty in using a single Q-matrix: Pathfinding problems usually require finding the shortest path between any two points in a graph. Since the graph can contain a large number of nodes and the required routes can vary a lot, representing all possible combinations in a single Q-matrix becomes challenging. This can lead to suboptimal solutions or even wrong results if the Q-values become hard to distinguish.
2) Balance between exploration and exploitation: In Q-learning, striking the right balance between exploration (trying new actions) and exploitation (using known actions) is critical to achieving better outcomes. However, in certain scenarios, such as when trying to find the shortest path between multiple nodes, the algorithm may end up making decisions that are overly guided by immediate rewards and not optimal overall.
3) Q-learning may lack information about the global structure of the graph: QL primarily learns from trial and error and does not leverage any prior knowledge of the graph structure, which can lead to potentially suboptimal decisions.

It is also important to consider the scale of the graph when choosing the algorithm in SPP problem. In scenarios where the graph is relatively small, RL might not be the most suitable approach, as dynamic programming (DP) or classical planning methods may provide better results. However, as the scale of the graph becomes substantially larger, making DP infeasible, RL may offer slightly better results than greedy algorithms due to its ability to incorporate experience from prior training. This allows RL to make more informed decisions and potentially find more efficient paths than a simple greedy algorithm.

E. Time and Memory Consuming

In terms of runtime and memory usage, We observed significant differences among the Dynamic Programming (DP), Q-Learning, and DQN algorithms.

For the Q-Learning algorithm, irrespective of the size of the graph, the agent can complete its learning process within approximately 10 seconds for a fixed number of epochs (set at 100,000). This fast performance is mainly due to the imposed limit on the number of steps the agent can take in each episode. The learning process thus finishes in a relatively short time frame, showcasing the algorithm's efficiency.

As for the Bidirectional Q-Learning algorithm, despite the fact that it deploys two agents (one starting from the beginning and the other from the end), there was no noticeable increase in the training time compared to unidirectional Q-Learning. The reason for this is that the two agents can operate concurrently, effectively halving the search space and maintaining a similar level of computational efficiency.

When it comes to the DQN algorithm, which incorporates neural networks into the learning process, the situation changes. The complexity of the neural network (specifically, the size of the input layer) is tied to the number of nodes in the graph. In our experiments conducted on a TESLA V100 card, each epoch took roughly one second for graphs with less than 100 nodes (i.e., when the input layer of the neural network had fewer than 300 units). However, as the number of nodes (and hence the input layer size) increased, the network training slowed considerably. In fact, for larger graphs, the training speed of the DQN could even fall below that of the DP algorithm.

Memory usage also varied among the algorithms. The DP algorithm consumed more memory than both Q-Learning and DQN. This can be attributed to the nature of the DP algorithm, which needs to keep track of the shortest path between all pairs of nodes. Consequently, its memory requirement grows quadratically with the number of nodes, making it less suitable for large-scale problems. On the other hand, the memory requirements for Q-Learning and DQN are relatively smaller as they only need to store the Q-values for each state-action pair, demonstrating their advantage in scalability.

V. Conclusion

In this peoject, we explored various DP and RL algorithms, including Floyd-Warshall, Q-Learning, Deep Q Networks (DQN), and Greedy algorithms, to solve the Shortest Path Problem (SPP). Dynamic Programming (DP) algorithms such as Floyd-Warshall consistently provided the best results in terms of path length, but these come with significant computational and memory demands, especially for larger graphs.

Reinforcement Learning (RL) algorithms like Q-Learning and DQN, while they did find paths, often resulted in paths significantly longer compared to DP. Despite this, RL algorithms still have certain utility in large-scale graphs, particularly when they can learn over time and adapt to changing environments. However, it is essential to balance the trade-off between computational efficiency and the quality of the solution.

For graphs of unknown scale and state, using a straightforward Greedy algorithm can be a feasible strategy. This is because they base decisions on local, immediate knowledge and do not require prior understanding of the entire graph.

Notably, introducing bidirectional search into both RL and Greedy algorithms significantly optimizes the discovered paths. Particularly in large graphs of 300 nodes, the total length of the path traversed by the agent was reduced by over tenfold compared to a standalone Q-Learning algorithm. Thus, bidirectional search approaches have shown to be a highly effective enhancement in such scenarios.

In conclusion, the choice of an appropriate algorithm for SPP largely depends on the characteristics of the graph, notably its size and whether its structure is known in advance. The integration of bidirectional search provides a substantial improvement in the path quality across different algorithmic strategies.

A. Supplementary Material

The code of Floyd-Warshall DP and Q-learning can be found in q-learning.ipynb, The code of DQN can be found in dqn.ipynb, the code for searching inside a graph with unknown node number and edges can be found in unknown_node_graph.ipynb.

References

[1] Z. Mammeri, "Reinforcement learning based routing in networks: Review and classification of approaches," *Ieee Access*, vol. 7, pp. 55 916–55 950, 2019.

[2] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 15, pp. 2–1, 2010.